Synchronization primitives

how low-level concurrency works

Outline

- Critical section example with a FIFO
- Case study: the spinlock
- Lock-free synchronization of the FIFO

Multiprocessing environments

Almost all consumer goods that resemble the traditional notion of a “computer” are multiprocessing.

Common parallel programming flavours:

- threads (inside one process)
- processes

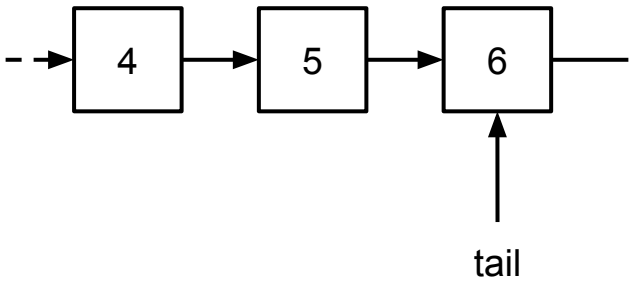
Multiprocessing environments

In multiprocessing environments, processes (threads) effectively execute simultaneously because of multiple CPU-s (cores) and context switching.

If they use a common resource, synchronization is needed.

How to corrupt a FIFO

A FIFO, or a queue, is a basic data structure where elements go in at the tail and come out at the head. The example is about the tail end.



FIFO in C

One FIFO element; the queue is a linked list of these:

```
struct elem {
  int data;
  struct elem *next;
};
```

FIFO in C

Adding to the tail looks like this (no synchronization here):

```
void add_elem(struct elem **tail,struct elem *e) {
   (*tail)->next = e;
   *tail = e;
}
```

Disassembly of `add_elem()`

```
mov   -0x8(%rbp),%rax    /* RAX<-tail pointer addr (stack)*/
mov   (%rax),%rax        /* RAX<-tail elem addr */
mov   -0x10(%rbp),%rdx   /* RDX<-addr of new elem (stack) */
mov   %rdx,0x8(%rax)     /* tail elem+8 bytes offset<-RDX */

mov   -0x8(%rbp),%rax    /* RAX<-tail pointer addr (stack)*/
mov   -0x10(%rbp),%rdx   /* RDX<-addr of new elem (stack) */
mov   %rdx,(%rax)       /* tail pointer<-addr of new elem */
```

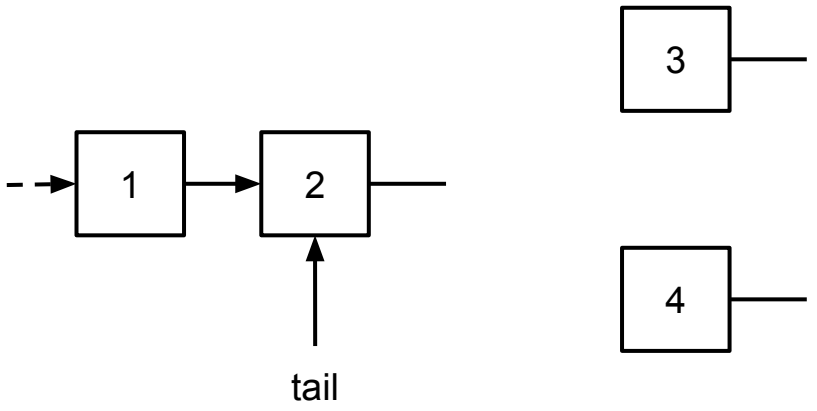
How to corrupt a FIFO

```
mov   -0x8(%rbp),%rax
mov   (%rax),%rax
mov -0x10(%rbp),%rdx
mov   %rdx,0x8(%rax)

mov   -0x8(%rbp),%rax
mov   -0x10(%rbp),%rdx
mov   %rdx,(%rax)
```

(red: t1, blue: t2)

New elements created. *t1* reads tail pointer.



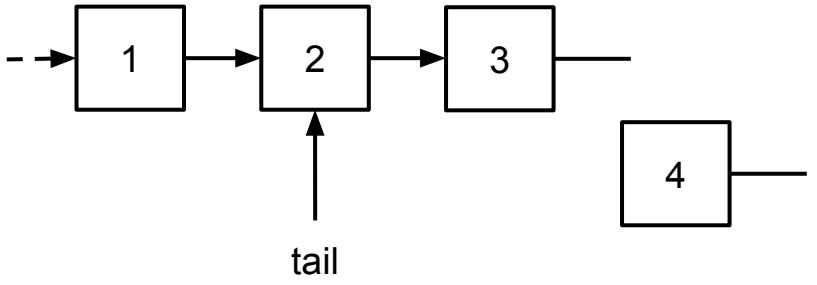
How to corrupt a FIFO

```
mov   -0x8(%rbp),%rax
mov   (%rax),%rax
mov -0x10(%rbp),%rdx
mov   %rdx,0x8(%rax)

mov   -0x8(%rbp),%rax
mov   -0x10(%rbp),%rdx
mov   %rdx,(%rax)
```

(red: t1, blue: t2)

t1 overwrites next pointer. *t2* reads tail pointer.



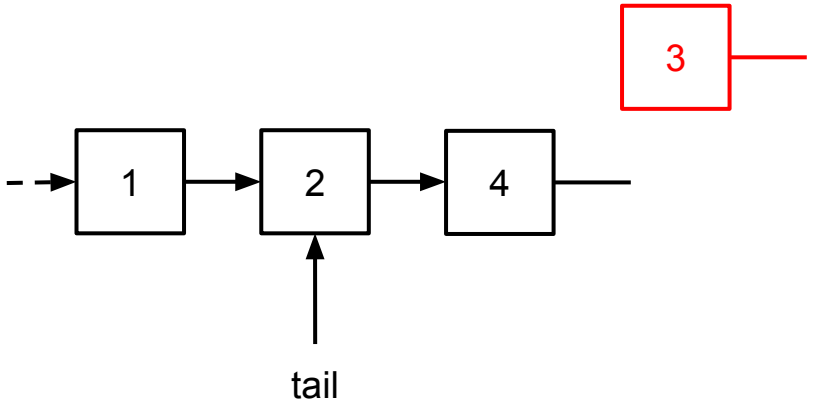
How to corrupt a FIFO

```
mov   -0x8(%rbp),%rax
mov   (%rax),%rax
mov -0x10(%rbp),%rdx
mov   %rdx,0x8(%rax)

mov   -0x8(%rbp),%rax
mov   -0x10(%rbp),%rdx
mov   %rdx,(%rax)
```

(red: t1, blue: t2)

t2 overwrites next pointer (same tail!). Orphan “3”.



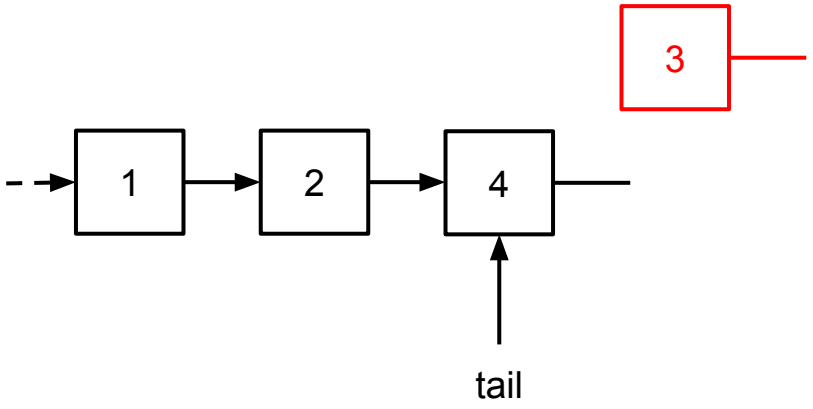
How to corrupt a FIFO

```
mov   -0x8(%rbp),%rax
mov   (%rax),%rax
mov -0x10(%rbp),%rdx
mov   %rdx,0x8(%rax)

mov   -0x8(%rbp),%rax
mov   -0x10(%rbp),%rdx
mov   %rdx,(%rax)
```

(red: t1, blue: t2)

t1 sleeps, *t2* updates tail pointer.



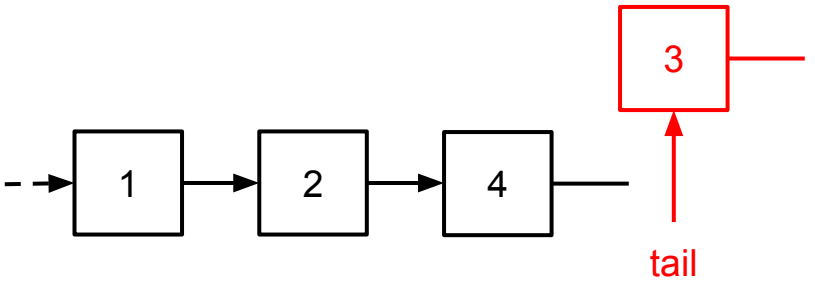
How to corrupt a FIFO

```
mov   -0x8(%rbp),%rax
mov   (%rax),%rax
mov -0x10(%rbp),%rdx
mov   %rdx,0x8(%rax)

mov   -0x8(%rbp),%rax
mov   -0x10(%rbp),%rdx
mov   %rdx,(%rax)
```

(red: t1, blue: t2)

t1 updates tail pointer. Queue split in two halves.



Synchronization

One thread must be able to complete the two pointer updates (the “critical section”) in our FIFO code before any other thread reads the tail pointer.

The obvious solution is to delay other threads that are attempting to enter the critical section.

Available methods

Many out-of-the box methods exist:

- IPC semaphores (UNIX-like)
- `pthreads` API (UNIX-like)
- Windows native API mutexes etc.
- Built in to languages: Java, Go, Ada
- libraries for C, Python etc (sometimes using the above methods internally)

The spinlock

The spin-wait loop is the basic building block of synchronization.

```
syn_var:
    .quad 0x0
spin_wait:
    lock btsq $0, syn_var /* read and set LSB of syn_var */
    jc spin_wait          /* was LSB already set? */
    /* critical section */
    movq $0, syn_var      /* clear LSB */
```

Basic spinlock flaws

- On a single CPU core, other threads make no progress during spinning.
- Spinning is considered a waste of resources as it does not do anything constructive.

Solution: yield the CPU to other threads after a very short spin.

Spinlock and caches

On multi-core cache coherent systems, all cores running the parallel program have the sync. variable in their cache.

- If the variable is written to, other cores must invalidate their entry
- then, a read will cause a cache miss, triggering a memory write AND read.

Spinlock and caches

By replacing most of the writes with reads, the spinlock will put less load on CPU interconnect and main memory.

```
spin_wait:
    cmpq $0, syn_var  /* syn_var == 0?  */
    jne spin_wait     /* if not, go read again */
    lock btsq $0, syn_var
    jc spin_wait
    /* critical section */

    movq $0, syn_var
```

Out of order execution

Reads are much faster than writes. Use the SSE2 `PAUSE` instruction to slow down reads, so the CPU has less work.

```
    cmpq $0, syn_var
    je try_lock
spin_wait:
    pause
    cmpq $0, syn_var
    jne spin_wait
try_lock:
    lock btsq $0, syn_var
    jc spin_wait
    /* critical section */
```

Spinlock and caches

Cache operates in larger units (64, 128 bytes).

Sync. variables and other data can end up on the same “cache lines”. Writing other data can cause sync. variables getting invalidated and vice versa, causing unnecessary cache coherence protocol traffic.

Alignment to cache line

Solution: synchronization variables placed alone in cache line-sized chunks.

```
struct syn_str { syn_var_t syn_var; };
void *p = malloc(sizeof(struct syn_str) + 127 );
syn_str *align_p = (syn_str *)((((ptrdiff_t) p) + 127) &\
  (ptrdiff_t) -128);
```

Practical spinlock

The next example contains the following optimizations:

- use `PAUSE` (via MM_PAUSE macro)
- Read before issuing an atomic operation
- Sleep to yield the CPU

```
int i;
struct timespec ts;
volatile syn_var_t *syn_var = someptr;

if(compare_and_swap(syn_var, 0, 1))
   return;
ts.tv_sec = 0;
ts.tv_nsec = SLEEP_NSEC;
for(;;) {
  for(i=0; i<SPIN_COUNT; i++) {
    MM_PAUSE
    if(!(*syn_var) && compare_and_swap(syn_var, 0, 1))
      return;
  }
  nanosleep(&ts, NULL);
  ts.tv_nsec += SLEEP_NSEC;
}
```

Compare and swap

The example used the compare-and-swap (CAS) atomic operation.

```
int CAS(addr, old, new) {  /* pseudocode */
  if(*addr == old) {
     *addr = new; return 1;
  } else
     return 0;
}
```

Lock-freedom

Lock-free data structures are an alternative to locking.

The idea is to not to block access to data, but rather modify it using careful protocols that do not permit it to become corrupted. This is quite difficult, but some practical algorithms exist.

Lock-freedom

Properties of lock-free algorithms

- do not suffer from deadlocks
- usually, a thread dying or stalling does not prevent others from working
- not necessarily faster that locking. This varies on case-by-case basis.

Lock-free FIFO

The introductory FIFO problem can also be solved with a lock-free algorithm.

The next slide shows a full listing based on the lock-free queue algorithm of Michael and Scott. Only adding to the queue is covered.

```
void add_elem(struct elem **tail, struct elem *e) {
  struct elem *t, *next;
  for(;;) {
    t = *tail;
    __sync_synchronize(); /* always re-read tail */
    next = t->next;
    __sync_synchronize(); /* preserve order here too */
    if(*tail != t) continue;
    if(next) {
      compare_and_swap(tail, t, next);
      continue;
    }
    if(compare_and_swap(&(t->next), NULL, e))
      break;
  }
  compare_and_swap(tail, t, e);
}
```

Analysis of the code

```
if(compare_and_swap(&(t->next), NULL, e))
  break;
```

This is where we want to get: the tail element's `next` pointer is NULL and we will chain our new element to it. However, this attempt alone is not good enough.

Analysis of the code

```
if(next) {
  compare_and_swap(tail, t, next);
  continue;
}
```

We cannot make progress if something is already linked to the tail element. And it’s not guaranteed that the tail pointer will be updated, so try to advance it.

Analysis of the code

```
t = *tail;
next = t->next;
if(*tail != t) continue;
```

This is how we knew that `next` even belonged to the tail element at the moment we read it: `t` points to the tail element before and after the read.

Analysis of the code

The tail pointer and its `next` field will be read over and over, until linking the element succeeds. Then we update the tail pointer. Failure here just means that some other thread updated it first.

```
compare_and_swap(tail, t, e);
```

Memory management

Lock-free algorithms frequently do not work without special memory management, because of the “ABA problem”.

Due to the cooperative nature, other threads may complete work with the data structures a thread is holding pointers to, then free and reuse them.

Memory management

```
if(*tail != t) continue;
if(next) { /* … assuming next == NULL
  tail element can move up queue and be freed
… */
if(compare_and_swap(&(t->next), NULL, e))
```

For example, the tail node can be reused for something else. As long as the offset of the `next` pointer happens to contain NULL, we still overwrite it.

Memory management

Solutions:

- reference counts
- powerful atomic operations found only in exotic hardware or on paper
- hazard pointers

Hazard pointers

- Keep a global array of “hazard” pointers vulnerable to ABA (or just freeing before use).
- If a structure is no longer needed, add it to a thread-local waiting list.
- Waiting list pointers not in the the hazard array may be freed.

Final thoughts

Typical synchronization problems have out-of-the box solutions (just need to be aware of them).

However, studying internals of low-level mechanisms grants deeper understanding of computers and programming.